Name of Student: Pushkar Sane			
Roll Number: 45		**Lab Assignment Number: 5**	
Title of Lab Assignment: To implement programs on OOP Concepts in python.			
DOP: 22-10-2023		**DOS: 26-10-2023**	
CO Mapped: **CO2**	**PO Mapped:** **PO5, PSO1**		**Signature:**

Practical No. 5

Aim: Implement programs on OOP Concepts in python.

1. Python Program to Create a Class and Compute the Area and the Perimeter of the Circle.
2. To Implement Multiple Inheritance in python.
3. To Implement a program with the same method name and multiple arguments.
4. To Implement Operator Overloading in python.
5. Write a program which handles various exceptions in python.

Description:

Object-Oriented Programming (OOP) is a programming paradigm that focuses on organizing code into objects, which are instances of classes. Python is an object-oriented programming language, and it supports the fundamental OOP concepts.

Here are the key OOP concepts in Python:

1. **Classes and Objects:**
    - A class is a blueprint or template for creating objects.
    - An object is an instance of a class.
    - Classes define attributes (data) and methods (functions) that the objects created from them will have.
    - Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def say_hello(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
# Creating objects from the class
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
# Accessing object attributes and methods
print(person1.name)   # Alice
person2.say_hello()   # Hello, my name is Bob and I am 25 years old.
```

2. **Inheritance:**
   - Inheritance allows you to create a new class that is based on an existing class. The new class inherits the attributes and methods of the base class.
   - Python supports single and multiple inheritance.
   - Example:

```
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id
    def study(self):
        print(f"{self.name} is studying.")
student = Student("Charlie", 20, "12345")
student.say_hello()     # Inherits from the Person class
student.study()         # Additional method specific to Student class
```

3. **Encapsulation:**
   - Encapsulation is the concept of restricting access to certain parts of an object or class, typically by using private and public access modifiers.
   - In Python, there are no true private variables or methods, but you can use naming conventions (e.g., prefixing with an underscore) to indicate privacy.
   - Example:

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance # Private attribute
    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
    def withdraw(self, amount):
        if amount > 0 and amount <= self._balance:
            self._balance -= amount
    def get_balance(self):
        return self._balance        # Public method to access balance
account = BankAccount(1000)
account.deposit(500)
account.withdraw(200)
print(account.get_balance())  # 1300
```

4. **Polymorphism**
   - Polymorphism allows objects of different classes to be treated as objects of a common base class.
   - It simplifies code by allowing you to work with objects generically without needing to know their specific types.
   - Example:

```
class Shape:
   def area(self):
      pass
   class Circle(Shape):
      def __init__(self, radius):
         self.radius = radius
   def area(self):
      return 3.14 * self.radius * self.radius
class Square(Shape):
   def __init__(self, side_length):
      self.side_length = side_length
   def area(self):
      return self.side_length * self.side_length
shapes = [Circle(5), Square(4)]
for shape in shapes:
   print(f"Area: {shape.area()}")
```

These are the core OOP concepts in Python. They help you create modular, reusable, and organized code by promoting the use of objects and classes.

5. **Overloading:**

   In Python, method overloading is not supported in the same way it is in some other programming languages, like Java or C++. Method overloading typically refers to defining multiple methods with the same name in a class but with different parameter lists. In Python, if you define multiple methods with the same name in a class, the last one defined will override any previous ones, and you can't call the overloaded method with different argument types as you can in languages that support method overloading.

Example:

```
class MyClass:
    def my_method(self, param1, param2=None):
        if param2 is not None:
            # Do something when both parameters are provided
            result = param1 + param2
        else:
            # Do something when only one parameter is provided
            result = param1
        return result
obj = MyClass()
result1 = obj.my_method(5)
result2 = obj.my_method(5, 10)
print(result1)   # Output: 5
print(result2)   # Output: 15
```

In the example above, the `my_method` function is defined with two parameters, but the second parameter has a default value of `None`. This allows you to call the method with one or two arguments, and the method behaves differently depending on the number of arguments provided.

Keep in mind that this approach is more Pythonic and flexible than traditional method overloading found in some other languages because it allows you to adapt the method's behavior at runtime. It's also easier to read and understand.

In Python, exceptions are used to handle errors and unexpected situations in your code. Here are some common exceptions in Python and examples of how they can occur:

6. **Exceptions in Python:**
    a. **Syntax Error:** Occurs when there is a syntax error in your code.
    Example:
    ```
    def some_function()
    ```

```
# Missing colon at the end of the function definition
print("Hello, World!")
```

b. **IndentationError:** Occurs when there is an issue with the indentation of your code.
   Example:
   ```
   if True:
   print("This line is not properly indented.")
   ```

c. **NameError:** Occurs when a local or global name is not found.
   Example:
   ```
   print(variable_that_does_not_exist)
   ```

d. **TypeError:** Occurs when you try to perform an operation on incompatible data types.
   Example:
   ```
   num = "5"
   result = num + 7 # Trying to add a string and an integer
   ```

e. **ValueError:** Occurs when a function receives an argument of the correct data type but an inappropriate value.
   Example:
   ```
   int("abc")  # Converting a non-numeric string to an integer
   ```

f. **IndexError:** Occurs when you try to access an index that is out of range in a sequence (e.g., a list or string).
   Example:
   ```
   my_list = [1, 2, 3]
   print(my_list[5])  # Accessing an index that doesn't exist
   ```

g. **KeyError:** Occurs when you try to access a dictionary key that doesn't exist.
   Example:
   ```
   my_dict = {"name": "John", "age": 30}
   print(my_dict["city"])  # Accessing a key that is not in the dictionary
   ```

h. **FileNotFoundError:** Occurs when you try to open a file that does not exist.
Example:
```
with open("nonexistent_file.txt", "r") as file:
    data = file.read()
```

i. **ZeroDivisionError:** Occurs when you attempt to divide by zero.
Example:
```
result = 5 / 0
```

j. **ImportError:** Occurs when there is an issue with importing a module or library.
Example:
```
import non_existent_module
```

k. **AttributeError:** Occurs when you try to access an attribute or method that does not exist.
Example:
```
class MyClass:
    def __init__(self, value):
        self.value = value
obj = MyClass(42)
print(obj.non_existent_attribute)
```

l. **Custom Exceptions:** You can also define and raise your own custom exceptions.
Example:
```
class MyCustomException(Exception):
    def __init__(self, message):
        super().__init__(message)
        raise MyCustomException("This is a custom exception.")
```

Handling exceptions allows you to gracefully manage errors in your code, preventing it from crashing and providing meaningful feedback to users or developers. You can use `try`, `except`, `finally`, and other control structures to manage exception handling in Python.

1. **Python Program to Create a Class and Compute the Area and the Perimeter of the Circle.**

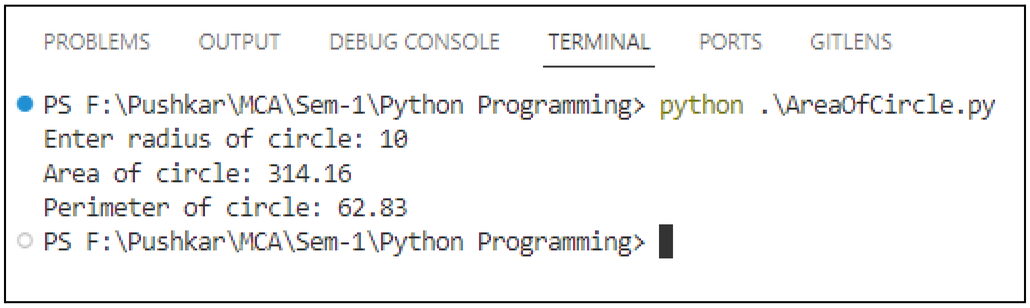
Code:

```
import math
class circle():
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * (self.radius ** 2)
    def perimeter(self):
        return 2 * math.pi * self.radius

r = int(input("Enter radius of circle: "))
obj = circle(r)
print("Area of circle:", round(obj.area(), 2))
print("Perimeter of circle:", round(obj.perimeter(), 2))
```

Conclusion:

In summary, this code defines a class circle that can calculate the area and perimeter of a circle given its radius. It takes user input for the radius, creates an instance of the circle class, and then calculates and displays the area and perimeter of the circle for that specific radius.

Output:



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\AreaOfCircle.py
Enter radius of circle: 10
Area of circle: 314.16
Perimeter of circle: 62.83
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

2. **To Implement Multiple Inheritance in python.**

Code:

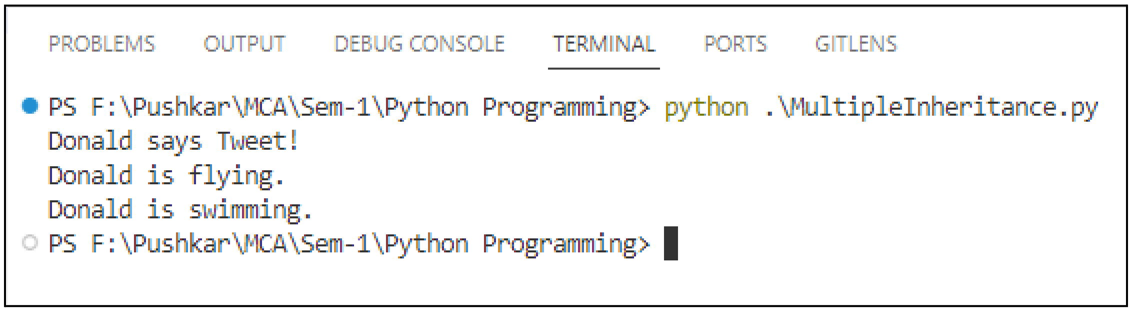
```
# Parent class 1
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        pass

# Parent class 2
class Flyable:
    def fly(self):
        pass

# Parent class 3
class Swimmable:
    def swim(self):
        pass

# Child class inheriting from Animal and Flyable
class Bird(Animal, Flyable):
    def speak(self):
        return f"{self.name} says Tweet!"
    def fly(self):
        return f"{self.name} is flying."

# Child class inheriting from Animal and Swimmable
class Fish(Animal, Swimmable):
    def speak(self):
        return f"{self.name} says Blub!"
    def swim(self):
        return f"{self.name} is swimming."
```

```
# Child class inheriting from Bird and Fish
class Duck(Bird, Fish):
    def __init__(self, name):
        super().__init__(name)

# Create instances of Duck and demonstrate multiple inheritance
donald = Duck("Donald")
print(donald.speak())
print(donald.fly())
print(donald.swim())
```

Conclusion:

This code demonstrates the concept of multiple inheritance in Python, where a class can inherit from multiple parent classes, allowing it to inherit and use the attributes and methods of all its parent classes. In this example, a "Duck" can both fly and swim due to its inheritance from both "Bird" and "Fish" parent classes.

Output:



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\MultipleInheritance.py
Donald says Tweet!
Donald is flying.
Donald is swimming.
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

3. **To Implement a program with the same method name and multiple arguments.**

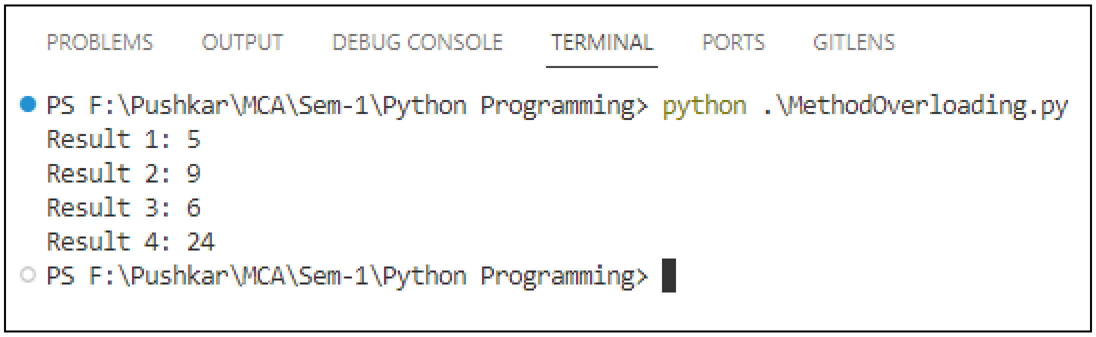
Code:

```
class MathOperations:
  def add(self, a, b, c=None):
    if c is not None:
      return a + b + c
    else:
      return a + b

  def multiply(self, a, b, c=None):
    if c is not None:
      return a * b * c
    else:
      return a * b

# Create an instance of the MathOperations class
math = MathOperations()

# Call the overloaded methods
result1 = math.add(2, 3)
result2 = math.add(2, 3, 4)
result3 = math.multiply(2, 3)
result4 = math.multiply(2, 3, 4)
print("Result 1:", result1)
print("Result 2:", result2)
print("Result 3:", result3)
print("Result 4:", result4)
```

Conclusion:

The code demonstrates how method overloading can be implemented in Python by defining methods with different numbers of parameters and using conditional statements to handle the variations in the number of arguments. This allows you to perform addition and multiplication operations with flexibility depending on the number of values provided as arguments.

Output:



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\MethodOverloading.py
Result 1: 5
Result 2: 9
Result 3: 6
Result 4: 24
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

4. **To Implement Operator Overloading in python.**

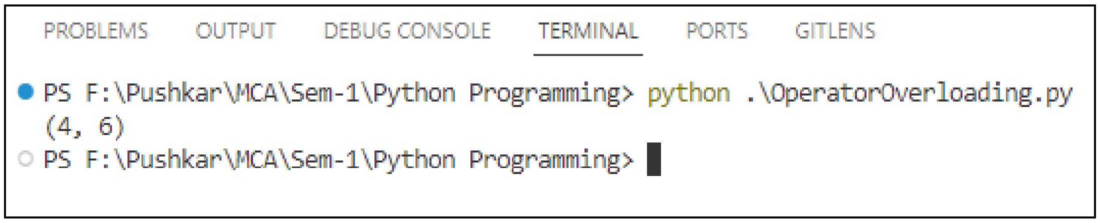
Code:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        if isinstance(other, Vector):
            # Add the x and y components of two vectors and return a new Vector
            return Vector(self.x + other.x, self.y + other.y)
        else:
            raise TypeError("Unsupported operand type for +: " + type(other).__name__)

    def __str__(self):
        return f"({self.x}, {self.y})"

# Create two Vector instances
v1 = Vector(1, 2)
v2 = Vector(3, 4)
# Use the overloaded + operator
result = v1 + v2
print(result) # Output: (4, 6)
```

Conclusion:

This code demonstrates how to create a Vector class that can perform vector addition using the + operator. The __add__ method is overloaded to handle this operation, making it convenient to work with vectors in a more natural and intuitive way.

Output:



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\OperatorOverloading.py
(4, 6)
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```

5. **Write a program which handles various exceptions in python.**

Code:

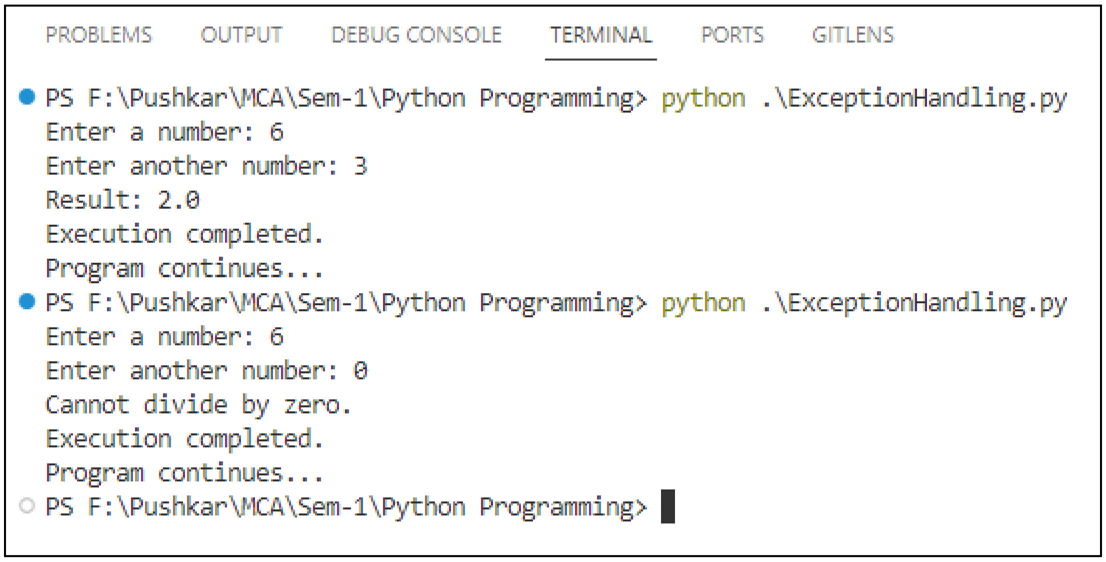
```
try:
    # Code that may raise exceptions
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))

    result = num1 / num2
    print(f"Result: {result}")

except ValueError:
    # Handle the ValueError exception (e.g., if user inputs a non-integer)
    print("Please enter valid integer values.")

except ZeroDivisionError:
    # Handle the ZeroDivisionError exception (e.g., if the second number is 0)
    print("Cannot divide by zero.")

except Exception as e:
    # Catch all other exceptions
    print(f"An error occurred: {e}")
```

```
finally:
    # This block is optional, and it will execute whether there's an exception or not.
    print("Execution completed.")
    # Rest of the program continues here...
    print("Program continues...")
```

Conclusion:

The code showcases how to gracefully handle exceptions and provides error messages for specific error scenarios while ensuring that the program doesn't terminate abruptly due to exceptions. The finally block allows you to execute cleanup or finalization code regardless of whether an exception occurred.

Output:



```
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\ExceptionHandling.py
Enter a number: 6
Enter another number: 3
Result: 2.0
Execution completed.
Program continues...
PS F:\Pushkar\MCA\Sem-1\Python Programming> python .\ExceptionHandling.py
Enter a number: 6
Enter another number: 0
Cannot divide by zero.
Execution completed.
Program continues...
PS F:\Pushkar\MCA\Sem-1\Python Programming>
```